Hotplug Memory Redux

Joel Schopp, Dave Hansen, & Mike Kravetz
IBM Linux Technology Center
jschopp@austin.ibm.com, haveblue@us.ibm.com, kravetz@us.ibm.com

Hirokazu Takahashi & IWAMOTO Toshihiro
VA Linux Systems Japan
taka@valinux.co.jp, iwamoto@valinux.co.jp

Yasunori Goto & KAMEZAWA Hiroyuki
Fujitsu
y-goto@jp.fujitsu.com, kamezawa.hiroyu@jp.fujitsu.com

Matt Tolentino
Intel
matthew.e.tolentino@intel.com

Bob Picco
HP
bob.picco@hp.com

Abstract

Memory Hotplug is one of the most anticipated features in the Linux Kernel. The purposes of memory hotplug are memory replacement, dynamic workload management, or Capacity on Demand of Partitioned/Virtual machines. In this paper we discuss the history of Memory Hotplug and the LinuxVM including mistakes made along the way and technologies which have already been replaced. We also discuss the current state of the art in Memory Hotplug including user interfaces, CONFIG_SPARSEMEM, the no bitmap buddy allocator, free area splitting within zones, and memory migration on PPC64, x86-64, and IA64. Additionally, we give a brief discussion on the overlap between Memory Hotplug and other areas including memory defragmentation and NUMA memory management. Finally, we gaze into the crystal ball to the future of Memory Hotplug.

1 Introduction

At the 2004 Ottawa Linux Symposium Andrew Morton had this to say in the keynote:

"Some features do tend to encapsulate poorly and they have their little sticky fingers into lots of different places in the code base. An example which comes to mind is CPU hot plug, and memory hot unplug. We may not, we may end up not being able to accept such features at all, even if they're perfectly well written and perfectly well tested due to their long-term impact on the maintainability of those parts of the software which they touch, and also to the fact that very few developers are likely to even be able to regression test them." [1]

It has been one year since that statement. Andrew Morton is a clever man who knows that the way to get developers to do something is to tell them it can't be done. CPU hot plug has been accepted[15]. The goal of this paper is to lay out how developers have been planning and coding to prove the memory half of that statement wrong.

2 Motivation

Memory hotplug was named for the ability to literally plug and unplug physical memory from a machine and have the Operating System keep running.

In the case of plugging in new physical memory the motivation is being able to expand system resources while avoiding downtime. The proverbial example of the usefulness is the slashdot effect. In this example a sysadmin runs a machine which just got slashdotted. The sysadmin runs to the parts closet, grabs some RAM, opens the case, and puts the RAM into the computer. Linux then recognizes the RAM and starts using it. Suddenly, Apache runs much faster and keeps up with the increased traffic. No downtime is needed to shutdown, insert RAM, and reboot.

Conversely, unplugging physical memory is usually motivated by physical memory failing. Modern machines often have the ability to recover from certain physical memory errors and to use those errors to predict that the physical memory is likely to have an unrecoverable error in the future. With memory hotplug the memory can be automatically disabled. The disabled memory can then be removed and/or replaced at the system administrator's convenience without downtime to the machine [31].

However, the ability to plug and unplug physical memory has been around awhile and nobody has previously taken it upon themselves to write memory hotplug for the Linux kernel. Fast forward to today and we have most major hardware vendors paying developers to write memory hotplug. Some things have changed; capacity upgrade on demand, partitioning, and virtualization all have made the resources assigned to an operating system much more fluid.

Capacity Upgrade On Demand came on the leading edge of this new wave. Manufacturers of hardware thought of a very clever and useful way to sell more hardware. The manufacturer would give users more hardware than they paid for. This extra unpaid for hardware would be disabled, and could be enabled if the customer later decided to pay for it. If the customer never decided to pay for it then the hardware would sit unused. Users got an affordable seamless upgrade path for their machines. Hardware manufacturers sold enough of the extra hardware they had already shipped they still made a profit on it. In business terms it was a win-win.

Without hotplug, capacity upgrades still require a reboot. This is bad for users who have to delay upgrades for scheduled downtime. The delayed upgrades are bad for hardware manufacturers who don't get paid for unupgraded systems. With hotplug the upgrades can be done without delay or downtime. It is so convenient that the manufacturers can even entice users with free trials of the upgrades and the ability to upgrade temporarily for a fraction of the permanent upgrade price.

The idea of taking a large machine and dividing up its resources into smaller machines is known as partitioning. Linux looks at a partition like it is a dedicated machine. This brings us back to our slashdotting example from physical hotplug. The reason that example didn't drive users to want hotplug was that it was only useful if there was extra memory in a closet somewhere and the system administrator could

open the machine while it was running. With partitioning the physical memory is already in the machine, it's just probably being used by another partition. So now hotplug is needed twice. Once to remove the memory from a partition that isn't being slashdotted and again to add it to a partition that is. The system administrator could even do this "hotplug" remotely from a laptop in a coffee house. Better yet management software could automatically decide and move memory around where it was needed. Because memory would be allocated more efficiently users would need less of it, saving them some money. Hardware vendors might even encourage selling less hardware because they could sell the management software cheaper than they sold the extra hardware it replaces and still make more money.

Virtualization then takes partitioning to the next level by removing the strict dependency on physical resources [10][17]. At first glance it would seem that virtualization ends the need for hotplug because the resources aren't real anyway. This turns out not to be the case because of performance. For example, if a virtual partition is created with 4GB of virtual RAM the only way to increase that to 256GB and have Linux be able to use that RAM is to hotplug add 252GB of virtual RAM to Linux. On the other side of the coin, if a partition is using 256GB of virtual RAM and whatever is doing the virtualizing only has 4GB of real honest-to-goodness physical RAM to use, performance will make it unusable. In this case the virtualization engine would want to hotplug remove much of that virtual RAM.

So there are a variety of forces demanding memory hotplug from hardware vendors to software vendors. Some want it for reliability and uptime. Others want it for workload balancing and virtualization.

Thankfully for developers it is also an interesting problem technically. There are lots of difficult problems to be overcome to make memory hotplug a success, and if there is one thing a developer loves it is solving difficult problems.

3 CONFIG_SPARSEMEM

3.1 Nonlinear vs Sparsemem

Previous papers[6] have discussed the concept of nonlinear memory maps: handling systems which have non-trivial relationships between the kernel's virtual and physical address spaces.

In 2004, Dave McCracken from IBM created a quite complete implementation of nonlinear memory handling for the hotplug memory project. As presented in[6], this implementation solved two problems: separating the `mem_map[]` into smaller pieces, and the nonlinear layout.

The nonlinear layout component turned out to be quite an undertaking. Its implementation required changing the types of some core VM macros: `virt_to_page()` and `page_to_virt()`. It also required changing many core assumptions, especially in boot-time memory setup code, which impaired other development. However, the component that separated the `mem_map[]`s turned out to be relatively problem-free.

The decision was made to separate the two components. Nonlinear layouts are not required by simple memory addition. However, the split-out `mem_map[]`s are. The memory hotplug plan has always been to merge hot-add alone, before hot-remove, to minimize code impact at one time. The `mem_map[]` splitting feature was named sparsemem, short for sparse memory handling, and the nonlinear portion will not be implemented until hot-remove is needed.

3.2 What Does Sparsemem Do?

Sparsemem has several of the same design goals as DISCONTIGMEM, which is currently in use in the kernel for similar purposes. Both of them allow the kernel to efficiently handle gaps in its address space. The normal method for machines without memory gaps is to have a `struct page` for each physical page of RAM in memory. If there are gaps from things like PCI config space, there are `struct page`'s, but they are effectively unused.

Although a simple solution, simply not using structures like this can be an extreme waste of memory. Consider a system with 100 1GB DIMM slots that support hotplug. When the system is first booted, only 1 of these DIMM slots is populated. Later on, the owner decides to hotplug another DIMM, but puts it in slot 100 instead of slot 2. This creates a 98GB gap. On a 64-bit system, each `struct page` is 64 bytes.

$$(\frac{98GB}{4096\frac{bytes}{page}}) * (64\frac{bytes}{struct\,page}) \approx 1.5GB$$

The owner of the system might be slightly displeased at having a **net loss** of 500MB of memory once they plug in a new 1GB DIMM. Both sparsemem and discontigmem offer an alternative.

3.3 How Does Sparsemem Work?

Sparsemem uses an array to provide different `pfn_to_page()` translations for each "section" of physical memory. The sections are arbitrarily sized and determined at compile-time by each specific architecture. Each one of these sections effectively gets its own, tiny version of the `mem_map[]`.

However, one must also consider the storage cost of such an array which must represent every possible physical address. Let's take PPC64 as an example. Its sections are 16MB in size and there are, today, systems with 1TB of memory in a single system. To keep future expansion in mind (and for easy math), assume that the limit is 16TB. This means 2^{20} possible sections and, with 1 64-bit `mem_map[]` pointer per section, that's 8MB of memory used. Even on the smallest (256MB) configurations, this amount is a manageable price to pay for expandability all the way to 16TB.

In order to do quick `pfn_to_page()` operations, the index into the large array of the page's parent section is encoded in `page->flags`. Part of the sparsemem infrastructure enables sharing of these bits more dynamically (at compile-time) between the `page_zone()` and sparsemem operations.

However, on 32-bit architectures, the number of bits is quite limited, and may require growing the size of the `page->flags` type in certain conditions. Several things might force this to occur: a decrease in the size of each section (if you want to hotplug smaller, more granular, areas of memory), an increase in the physical address space (very unlikely on 32-bit platforms), or an increase in the number of consumed `page->flags`.

One thing to note is that, once sparsemem is present, the NUMA node information no longer needs to be stored in the `page->flags`. It might provide speed increases on certain platforms and will be stored there if there are unused bits. But, if there are inadaquate unused bits, an alternate (theoretically slower) mechanism is used; `page_zone(page)->zone_pgdat->node_id`.

3.4 What happens to Discontig?

As was noted earlier sparsemem and discontigmem have quite similar goals, although quite

different implementations. As implemented today, sparsemem replaces DISCONTIGMEM when enabled. It is hoped that SPARSEMEM can eventually become a complete replacement as it becomes more widely tested and graduates from experimental status.

A significant advantage sparsemem has over DISCONTIGMEM is that it's completely separated from CONFIG_NUMA. When producing this implementation, it became apparent that NUMA and DISCONTIG are often confused.

Another advantage is that sparse doesn't require each NUMA node's ranges to be contiguous. It can handle overlapping ranges between nodes with no problems, where DISCONTIGMEM currently throws away that memory.

Surprisingly, sparsemem also shows some marginal performance benefits over DISCONTIGMEM. The base causes need to be investigated more, but there is certainly potential here.

As of this writing there are ports for sparsemem on i386, PPC64, IA64, and x86_64.

4 No Bitmap Buddy Allocator

4.1 Why Remove the Bitmap?

When memory is hotplug added or removed, memory management structures have to be reallocated. The buddy allocator bitmap was one of these structures.

Reallocation of bitmaps for Memory Hotplug has the following problems:

- Bitmaps were one of the things which assumed that memory is linear. This assumption didn't fit SPARSEMEM and Memory Hotplug.
- For resizing, physically contiguous pages for new bitmaps were needed. This increased possibility of failure of Memory Hotplug because of difficulty of large size page allocation.
- Reallocation of bitmaps is complicated and computationally expensive

For Memory Hotplug, bitmaps presented a large obstacle to overcome. One proposed solution was dividing and moving bitmaps from zones to sections as was done with memmaps. The other proposed solution, eliminating bitmaps altogether, proved simpler than moving them.

4.2 Description of the Buddy Allocator

The buddy allocator is an memory allocator which coalesces pages into groups of 2^X length. X is usually 0-10 in Linux. Pages are coalesced into a group of length of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024. X is called an "order". Only a head page of a buddy is linked to `free_area[order]`.

This grouping is called a buddy. A pair of buddies in order X, which are length of 2^X, can be coalesced into a buddy of $2^{(X+1)}$ length. When a pair of buddies can be coalesced in order X, offset of 2 buddies are $2^{(X+1)} * Y$ and $2^{(X+1)} * Y + 2^X$. Hence, another buddy of a buddy in order X can be calculated as (Offset of a buddy) XOR $(1 << (X))$.

For example, page 4 (0x0100) can be coalesced with page 5 (0x0101) in order 0, page 6 (0x0110) in order 1, page 0 (0x0000) in order 2.

4.3 Bitmap Buddy Allocator

The role of bitmaps was to record whether a page's buddy in a particular order was free or

not. Consider a pair of buddies at $2^{(X+1)} * Y$ and $2^{(X+1)} * Y + 2^X$.

When `free_area[X].bitmap[Y]` was 1, one of the buddies was free. So `free_pages()` can determine whether a buddy can be coalesced or not from bitmaps. When both buddies were freed, they were coalesced and `free_area[X].bitmap[Y]` set to 0.

4.4 No Bitmap Buddy Allocator

When it comes to the no bitmap buddy allocator, instead of recording whether a page has its buddy or not in a bitmap, the free buddy's order is recorded in `memmap`. The following expression is used to check a buddy page's status:

```
page_count(page) == 0 &&
PG_private is set &&
page->private == X
```

The three elements that make up this expression are:

- When `page_count(page) == 0`, page is not used.
- Even if `page_count(page) == 0`, it's not sure that the page is linked to the free area. When a page is linked to the free area, `PG_private` is set.
- When `page_count(page)==0 && PG_private` is set, `page->private` indicates its order.

Here, offset of an another buddy of a buddy in order X can be calculated as (Offset of page) XOR 2^X. The following code is the core of no bitmap buddy allocator's coalescing routine:

```
struct page *base = zone->zone_mem_map;
int page_idx = page - base;
while ( order < MAX_ORDER ) {
  int buddy_idx = page_idx ^ (1 << order);
  struct page *buddy = base + buddy_idx;
  if (!(page_count(buddy) == 0 &&
        PagePrivate(buddy) &&
        buddy->private == order))
              break;
  remove buddy from zone->free_area[order]
  ClearPagePrivate(buddy);
  if (buddy_idx < page_idx)
        page_idx = buddy_idx;
  order++;
}
page = page_idx + base;
SetPagePrivate(page);
page->private = order;
link page to zone->free_area[order]
```

There is no significant performance difference either way between bitmap and no bitmap coalescing.

With SPARSEMEM, `base` in the above code is removed and following the relative offset calculation is used. Thus, the buddy allocator can manage sparse memory very well.[1]

```
page_idx = pfn_to_page(page);
buddy_idx = page_idx ^ (1 << order);
buddy = page + (buddy_idx - page_idx);
```

5 Free Area Splitting Within Zones

The buddy system provides an efficient algorithm for managing a set of pages within each zone [7][16][18][11]. Despite the proven effectiveness of the algorithm in its current form as used in the kernel, it is not possible to aggregate a subset of pages within a zone according to specific allocation types. As a result, two physically contiguous page frames (or sets of page frames) may satisfy allocation requests that are drastically different. For example, one page frame may contain data that is only temporarily used by an application while the other

[1] SPARSEMEM guarantees that memmap is contiguous at least up to MAX_ORDER.

is in use for a kernel device driver. While this is perfectly acceptable on most systems, this scenario presents a unique challenge on memory hotplug systems due to the variances in reclaiming pages that satisfy each allocation type.

One solution to this problem is to explicitly manage pages according to allocation request type. This approach avoids the need to radically alter existing page allocation and reclamation algorithms, but does require additional structure within each zone as well as modification of the existing algorithms.

5.1 Origin of Page Allocation Requests

Memory allocations originate from two distinct sources—user and kernel requests. User page allocations typically result from a write into a virtual page in the address space of a process that is not currently backed by physical memory. The kernel responds to the fault by allocating a physical page and mapping the virtual page to the physical page frame via page tables. However, when the system is under memory pressure, user pages may be paged out to disk in order to reclaim physical page frames for other higher priority requests or tasks. The algorithms and techniques used to accomplish this function constitute much of the virtual memory research conducted to date[22][23][24][25][26][27].

In Linux, user level allocations may be satisfied from pages contained in any zone, although they are preferably allocated from the HIGHMEM zone if that zone is employed by the architecture. This is reasonable considering these pages are not permanently mapped by the kernel. Architectures that do not employ the HIGHMEM zone direct user level allocations to one of the other two zone types, NORMAL or DMA. Unlike user allocations, kernel allocations must be satisfied from memory that is permanently mapped in the virtual address space. Once a suitable zone is chosen, an appropriately sized region is plucked from the respective free area list via the buddy algorithm without regard for whether it satisfies a kernel allocation or a user allocation.

5.2 Distinguishing Page Usage

During a page allocation, attributes are provided to the memory allocation interface functions. Each attribute provides a hint to the allocation algorithm in order to determine a suitable zone from which to extract pages; however, these hints are not necessarily provided to the buddy system. In other words, the region from which the allocation is satisified is only determined at a zone granularity. On systems such as PPC64 this may include the entirety of system memory! In order to enable the distinction of user allocation from kernel allocations within a zone, additional flags that specify whether the region must be provided to the buddy algorithm. These flags include:

- User Reclaimable
- Kernel Reclaimable
- Kernel Non-Reclaimable

Using these flags, the buddy allocation algorithm may further differentiate between page allocations and attempt to maintain regions that satisfy similar allocations and more significantly, have similar presence requirements.

5.3 Multiple Free Area Lists

Existing kernels employ one set of free area lists per zone as shown in figure1. In order

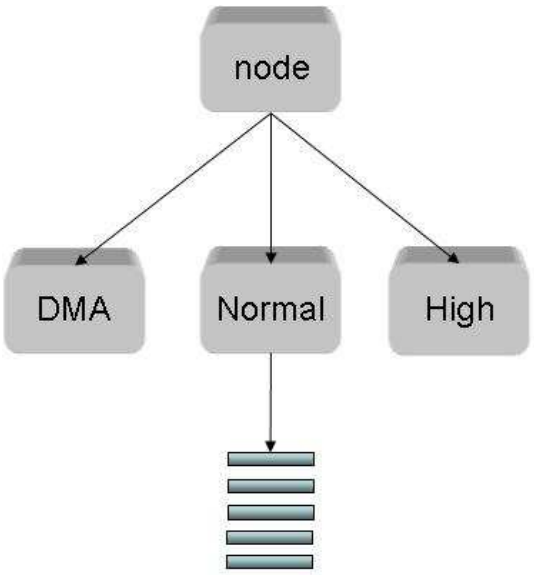

Figure 1: Existing free area list structure

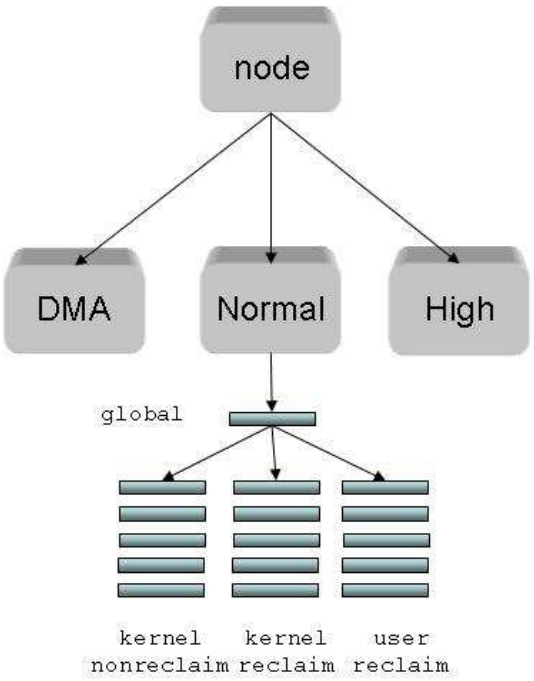

Figure 2: New free area list structure for fragmentation

to explicitly manage the user versus kernel distinction of memory with zones, multiple sets of free area lists are used within each zone, specifically one set of free area lists per allocation type. The basic strategy of the buddy algorithm remains unchanged despite this modification. Each set of free area lists employs the exact same splitting and coalescing steps during page allocation and reclamation operations. Therefore the functional integrity of the overall algorithm is maintained. The novelty of the approach involves the decision logic and accounting involved in directing allocations and free operations to the appropriate set of free area lists.

Mel Gorman posted a patch that implements exactly this approach in an attempt to minimize external memory fragmentation, a consistent issue with the buddy algorithm [19][8][7][11][16]. This approach introduces a new global free area list with MAX_ORDER sized memory regions and three new free area lists of size MAX_ORDER-1 as depicted in figure 2 below.

Although Mel's approach provides the basic infrastructure needed by memory hotplug, additional structure is required. In addition to the set of free area lists for each allocation type, an additional global free area list for contiguous regions of MAX_ORDER size is also maintained as depicted in figure three. The addition of this global list enables accounting for MAX_ORDER sized memory regions according to the capability to hotplug the region. Thus, during initialization, memory regions within each zone are directed to the appropriate global free area list based on the potential to hotplug the region at a later time. This translates directly to the type of allocation a page satisfies. For example, many kernel pages are pinned in memory and will never be freed. Hence, these pages will be obtained from the global *pinned* list. On the other hand nearly every user page may be reclaimed, so these pages will be obtained from the global *hotplug* list.

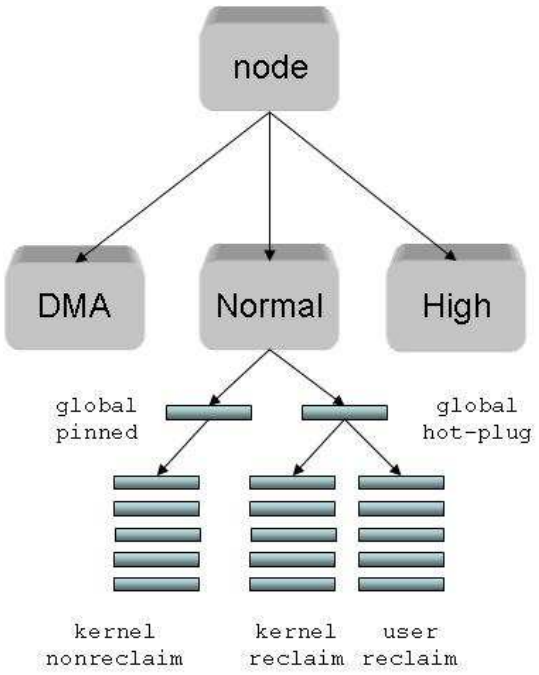

Figure 3: New free area list structure for memory hotplug

5.4 Memory Removal

The changes detailed in this section enable the isolation of sets of pages according to the type of allocation they may satisfy. Because user pages are relatively easy to reclaim, those page allocations will be directed to the regions that are maintained in the global *hotplug* free area list. During boot time or during a memory hot-add operation, the system firmware details which regions may be removed at runtime. This information provides the context for initializing the global *hotplug* list. As pages are allocated thus depleting the user and kernel reclaimable free area lists, additional MAX_ORDER regions may be derived from the global *hotplug* list. Similarly, the global *pinned* list provides pages to the kernel non-reclaimable lists upon depletion of available pages of sufficient size.

Because pages that may be hot-removed at runtime are isolated such that they satisfy user or kernel reclaimable allocations, memory removal is possible. This was not previously possible with the existing buddy algorithm due to the likelihood that a pinned kernel non-reclaimable page could be located within the range to be removed. Thus, kernels that only employ a subset of the potential zones may support hot-remove transparently.

5.5 Configurability

While satisfying allocation requests from discrete memory regions according to allocation type does enable removal of memory within zones, there is still the potential for one type of allocation to run out of pages due to the assignment of pages to each global free list. Mel dealt with this issue for the fragmentation problem by allowing allocations to *fallback* to another free area list should one become depleted[19].

While this is reasonable for most systems, it compromises the capability to remove memory should a non-reclaimable kernel allocation be satisfied by some set of pages in a hotplug region. As this type of fallback policy decision largely depends on the intended use of the system, one approach is to allow for the fallback decision logic to be configured by the system administrator. Therefore, systems that aren't likely to need to remove memory, even though the functionality is available, may allow the fallback to occur as the workload demands. Other systems in which memory removal is more critical may disable the fallback mechanism, thus preserving the integrity of hotplug memory regions.

6 Memory migration

6.1 Overview

In memory hotplug removal events, all the pages in some memory region must be freed in a timely fashion, while processes are running as usual. Memory migration achieves this by blocking page accesses and moves page contents to new locations.

Although using `shrink_list()` function—which is the core of `kswapd`—sounds simpler, it cannot reliably free pages and causes many disk I/Os. Additionally, the function cannot handle pages which aren't associated with any backing stores. Pages on a ramdisk are an example of this. The memory migration is designed to solve these issues. Page accesses aren't a problem because they are blocked, while `shrink_list()` cannot process pages that are being accessed. Unlike `shrink_list()`, most dirty pages can be processed without writing them back to disk.

6.2 Interface to Migrate Page

To migrate pages, create a list of pages to migrate and call the following function: `int try_to_migrate_pages(struct list_head *page_list)`

It returns zero on success, otherwise sets page_list to a list of pages that cannot migrate and returns a non-zero value. Callers must check return values and retry failed pages if necessary.

This function is primarily for memory hotplug remove, but also can be used for memory defragmentation (see Section 8.1) or process migration (see Section 8.2.3).

6.3 How does the memory migration work?

A memory migration operation consists of the following steps. The operation of anonymous pages is slightly different.

1. lock `oldpage`, which is the target page
2. allocate and lock `newpage`
3. modify `oldpage` entry in `page_mapping(oldpage)->page_tree` with `newpage`
4. invoke `try_to_unmap(oldpage, virtual_address_list)` to unmap `oldpage` from the process address spaces.
5. wait until `!PageWriteback(oldpage)`
6. write back `oldpage` if `oldpage` is dirty and `PagePrivate(oldpage)` and no file system specific method is available
7. wait until `page_count(oldpage)` drops to 2
8. `memcpy(newpage, oldpage, PAGE_SIZE)`
9. make `newpage` up to date
10. unlock `newpage` to wakeup the waiters
11. free `oldpage`

The key is to block accesses to the page under operation by modifying the `page_tree`. After the `page_tree` has been modified, no new access goes to `oldpage`. The accesses are redirected to `newpage` and blocked until the data is ready because it is locked and isn't up to date (Figure 4).

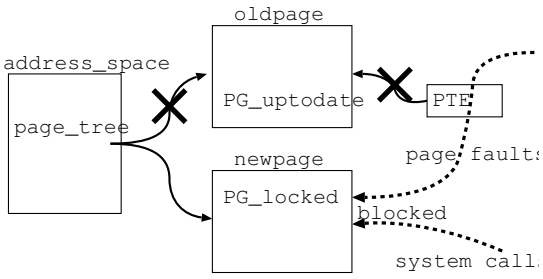

Figure 4: `page_tree` rewrite and page accesses

To handle `mlock()`ed pages, `try_to_unmap()` now takes two arguments. If the second argument is non-NULL, the function unmaps `mlock()ed` pages also and records unmapped virtual addresses, which are used to reestablish the PTEs when the migration completes.

Because the direct I/O code protects target pages with incremented `page_count`, memory migration doesn't interfere with the I/O.

In some cases, a memory migration operation needs to be rolled back and retried later. This is a bit tricky because it is likely that some processes have already looked up the `page_tree` and are waiting for its lock. Such processes need to discard `newpage` and look up the `page_tree` again, as `newpage` is now invalid.

6.3.1 Anonymous Memory Migration

The memory migration depends on `page_tree` lists of inodes, while anonymous pages may not correspond to any of them. This structure is strictly required to block all accesses to pages on it during migration.

Therefore, anonymous pages should be moved into the swap-cache prior to migrating them. After that these pages are placed in the `page_tree` of `swapper_space`, which manages all pages in the swap-cache. These pages can be migrated just like pages in the page-cache without any disk I/Os.

The important issue of systems without swap devices remains. To solve it, Marcelo Tosatti has proposed the idea of "migration cache"[2] and he's working on its implementation. The migration cache is very similar to the swap-cache except it doesn't require any swap devices.

6.4 Keeping Memory Management Aware of Memory Migration

The memory migration functionality is designed to fit the existing memory management semantics and most of the code works without a modification. However, the memory management code should satisfy the following rules:

- Multiple lookups of a page from its `page_tree` should be avoided. If a kernel function looks up a `page_tree` location multiple times, a memory migration operation can rewrite the `page_tree` in the meanwhile. When such a `page_tree` rewrite happens, it usually results in a deadlock between the kernel function and the memory migration operation. The memory migration implements a timeout mechanism to resolve such deadlocks, but it is preferable to remove the possibility of deadlocks by avoiding multiple `page_tree` lookups of the same page. Another option is to use non-hotremovable memory for such pages.
- The pages which may be grabbed for an unpredictably long time must be allocated from non-hotremovable memory,

even though it may be in the page-cache or anonymous memory. For instance, pages used as ring buffers for asynchronous input/output (AIO) events are pinned not to be freed.

- For a swap-cache page, its PG_swapcache flag bit needs checking after obtaining the page lock. This is due to how the memory migration is implemented for swap-cache pages and not directly related to unwinding. Such a check code is added in `do_swap_page()`.

- Functions that call `lock_page()` must be aware of the unwinding of memory migration. Basically, a page must be checked if it is still valid after every `lock_page()` call. If it isn't, one has to restart the operation from looking up the `page_tree` again. A good example of such restart is in `find_lock_page()`.

6.5 Tuning

6.5.1 Implementing File System Specific Methods for Memory Migration

Memory migration works regardless of file systems in use. However, it is desirable that file systems which are intensively used implement the helper functions which are described in this subsection.

There are various things that refer to pages, and some of these references need time consuming operations such as disk I/Os to complete in order for the reference to be dropped. This subsection focuses on one of these—the handling of dirty buffer structures pointed by `page->private`.

When a dirty page is associated with a buffer, the page must be made clean by issuing a write-back operation and the buffer must be freed unless there is an available file system specific operation defined.

The above operation can be too slow to be practical because it has to wait a writeback I/O completion. A file system specific operation can be defined to avoid this problem by implementing the `migrate_page()` method in the `address_space_operations` structure.

For example, the `buffer_head` structures that belong to ext2 file systems are handled by the `migrate_page_buffer` function. This function enables page migration without write-back operations by having `newpage` to take over the `buffer_head` structure pointed by `page->private`. It is implemented as follows:

- Wait until the `page_count` drops to the prescribed value (3 when the `PG_private` page flag is set, 2 otherwise). While waiting, issue the `try_to_unmap()` function calls.

- If the `PG_private` flag is set, process the `buffer_head` structure by calling the `generic_move_buffer()` function. The function waits until the `buffer_count` drops and the buffer lock is released. Then, it has `newpage` to take over the buffer_head structure by modifying `page->private`, `newpage->private` and the `b_page` member in the `buffer_head` structure. To adjust `page_count` due to the `buffer_head` structure, increment the `page_count` of `newpage` by one and decrement the one of `page` by one.

- At this point, the `page_count` of page is 2 regardless of the original state of the `PG_private` flag.

6.5.2 Combination With `shrink_list()`

Memory pressure caused by memory migration can be reduced. This memory pressure can cause reclaim of pages as replacements. Inactive pages are not worth migrating when the resultant migration causes other valid pages be reclaimed. This undesirable effect perturbs the LRUness of pages reclaimed. It would be preferable to just release these pages without migrating them.

The current implementation[3] invokes `shrink_list()` to release inactive pages and moves only active pages to new locations in case of memory hotplug removal.

6.6 Hugetlb Page Migration.

Due to certain workloads like databases and high performance computing (HPC) large page capability is critical for good performance. Because these pages are so critical to these workloads it follows that page migration must support migration of large pages to be widely used.

6.6.1 Interface to Migrate Hugetlb Pages

The prototype[5] interface for hugetlb migration seperates normal page migration from huge page migration.

When a caller notices the page needing to be migrated is a hugetlb page, it has to pass the page to `try_to_migrate_hugepage()`, migrating it without any system freeze or any process suspension.

6.6.2 Design of hugetlb page migration

The migration can be done in the same way for normal pages, using the same memory migration infrastructure. Luckily, it's not so hard to implement because Linux kernel manages large pages—often called hugetlb pages—via the pseudo file system known as hugetlbfs.

Linux kernel handles them in a similar manner as it handles normal pages in the page-cache. It inserts each of them into the `page_tree` of the associated inode in hugetlbfs and maps them into process address spaces using `mmap()` system call.

There is one additional requirement for migration of large pages. Demand paging against hugetlb pages must be blocked, with all accesses via process address spaces to pages under migration blocked in a pagefault handler until the migration is completed.

Therefore, the hugetlb page management related to demand paging feature has to be enhanced as follows:

- A pagefault handler for hugetlb pages must be implemented. The implementation[4] Chen, Kenneth W and Christoph Lameter are working on can be used with some modification, making the processes block in the pagefault handler if the page is locked. This is similar to what the pagefault handler for normal pages does.
- The function `try_to_unmap()` must be able to handle hugetlb pages to unmap them from process address spaces. This means `objrmap`—the object-based reverse mapping VM—also has to be introduced so that page table entries associated with any pages can be found easily.

Another interesting topic is hugetlb page allocation, which is almost impossible to do dynamically. Physically contiguous memory allo-

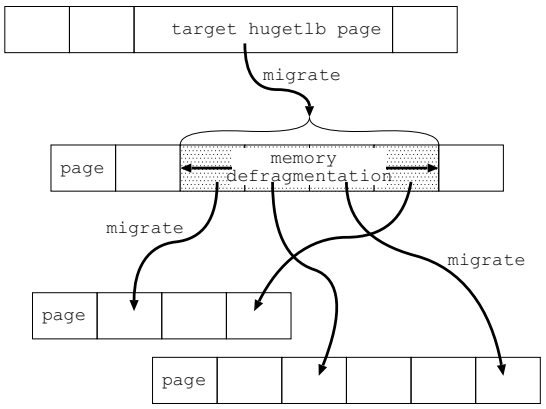

Figure 5: `hugetlb` page migration

cation is one of the well known issues remaining to be solved. The current hugetlb page management chooses the approach that all pages should be reserved at system start-up time.

Despite its current state, hugetlb page migration can not continue to use this approach. On demand allocation is strictly required. Fortunately, this is going to be solved with "Memory defragmentation" (see Section 8.1). Marcelo Tosatti is working on "Free area splitting within zones" effort (see section 5).

6.6.3 How `hugetlb` Page Migration Works

There really isn't much difference between hugetlb page migration and normal page migration. The following is the algorithm flow for this migration.

1. Allocate a new hugetlb page from the page allocator also known as the buddy allocator. This may require memory defragmentation to make a sufficient contiguous range (figure 5).
2. Lock the newly allocated page and keep it non-uptodate, without the `PG_uptodate` flag on it.
3. Replace a target hugetlb page with the new page on `page_tree` of the corresponding inode in hugetlbfs.
4. Unmap the target page from the process address spaces, clearing all page table entries mapping it.
5. Wait until all references on the target page are gone.
6. Copy from the target page to the new page.
7. Make the new page uptodate, setting the `PG_uptodate` flag on it.
8. Release the target page into the page allocator directly.
9. Unlock the new page to wake up all waiters.

6.7 Restriction

Under some rare situations, pages cannot migrate, and making those migrations functional would require too much code to be practical.

- NFS page-cache may have a non-responding NFS server. NFS I/O requests cannot complete if the server isn't responding. The pages with such outstanding NFS I/O requests cannot migrate. It is technically possible to handle this situation by updating all the references to an oldpage with ones to a newpage, but the code modification would be very large and probably not maintainable.
- Page-cache of which the file is used by `sendfile()` are also problematic. When a page-cache page is used by `sendfile()`, its page_count is kept raised until corresponding TCP packets are ACKed. This becomes a problem when a connection peer doesn't read data from the TCP connection.

- RDMA client/server memory use may also be an issue but further investigation is required.

6.8 Future Work

Currently, nonlinear mmaped pages[2] cannot migrate as `try_to_unmap()` doesn't unmap such pages. This must be addressed.

All file systems should have their own `migrate_page()` method. This will help performance considerably as the filesystems can make more intelligent decisions about their own data.

Kernel memory should be migratable too. A first approach would be migrating page table pages which consume significant memory. This migration should be reasonably straightforward.

7 Architecture Implementation Specifics

Memory hotplug has been implemented on many different architectures. Each of these architectures have unique hardware and consequently do memory management in different ways. They each present unique challenges and solutions that should be of interest to future implementators on the other architectures that currently don't support memory hotplug. Additionally, those whose architectures are already covered can better understand their own architectures by comparing them side by side with others.

[2]With `remap_file_pages()` system call, several pieces of a file can be mapped into one contiguous virtual memory.

7.1 PPC64 Implementation

The PPC64 architecture is perhaps the most mature with respect to the support of memory hotplug. This is because there are other operating systems that currently support memory hotplug on this architecture.

7.1.1 Logical Partition Environment

Operating Systems running on PPC64 operate in a Logical Partition (LPAR) of the machine. These LPARs are managed by a underlying level of firmware known as the hypervisor. The hypervisor manages access to the actual underlying hardware resources. It is possible to dynamically modify the resources associated with an LPAR. Such dynamically modifiable LPARS are known as Dynamic LPARS (DLPARs)[29].

Memory is one of the resources that can be dynamically added to or removed from a DLPAR on PPC64. In a PPC64 system, physical memory is divided into memory blocks that are then assigned to LPARs. The hypervisor performs remapping of real physical addresses to addresses that are given to the LPAR[3] These memory blocks with remapped addresses appear as physical memory to the Operating Systems in the LPAR. When an OS is started on an LPAR, the LPAR will have a set of memory blocks assigned to it. In addition, memory blocks can be added or removed to the LPAR while the OS is active. The size of memory blocks managed by the hypervisor is scaled based on the total amount of physical memory in the machine. The minimum size block

[3]To Linux the addresses given to it are considered physical addresses, but they are not in actuality physical addresses. This causes no end of confusion in developers conversations because developers get confused over what is a virtual address, physical address, remapped address, etc.

is 16MB[4]. As a result, the default SPARSEMEM section size for PPC64 is a relatively small 16MB.

7.1.2 Add/Remove Operations

On PPC64, the most common case of memory hotplug is not expected to be the actual addition or removal of DIMMs. Rather, memory blocks will be added to or removed from a DLPAR by the hypervisor. These add or remove operations are initiated on the Hardware Management Console (HMC). When memory is added to an LPAR, the HMC will notify a daemon running in the OS of the desire to add memory blocks. The daemon in turn makes a special system call that results in calls being made to the hypervisor. The hypervisor then makes additional memory blocks available to the OS. As part of the special system call processing, the physical address[5] of these new blocks is obtained. With the physical address known, scripts called via the daemon use the sysfs memory hotplug interface to create new memory sections associated with the memory blocks.

For memory remove operations, the HMC once again contacts the daemon running in the OS. The OS then executes a script that uses the sysfs interfaces to offline a memory section. Once a section is offlined, a special system call is made that results in calls to the hypervisor to isolate the memory from the DLPAR.

[4] 256MB is a more typical minimum block size. On some machines the user can actually change the minimum block size the machine will use

[5] This is not the real physical address, but the remapped address that Linux thinks is a real physical address

7.1.3 Single Zone and Defragmentation

PPC64 makes minimal use of memory zones. This is because DMA operations can be performed to any memory address. As a result, only a single DMA zone is created on PPC64 and no HIGHMEM or NORMAL zones. Of course, there may be multiple DMA zones (one per node) on a NUMA architecture. Having a single zone makes things simpler but it does nothing to segregate memory allocations of different types. For example, on architectures that support HIGHMEM, allocations for user pages mostly come from this zone. Having multiple zones provides a very basic level of segregation of different allocation types. Since we have no such luxury on PPC64, we must employ other methods to segregate allocation types. The memory defragmentation work done by Mel Gorman is a good starting point for this effort[19]. Mel's work segregates memory allocations on the natural MAX_ORDER PAGE size blocks managed by the page allocator. Because PPC64 has a relatively small default section size of 16 MB, it should be possible to extend this concept in an effort to segregate allocations to segment size blocks.

7.1.4 PPC64 Hypervisor Functionality

The PPC64 hypervisor provides functionality to aid in the removal of memory sections. The H_MIGRATE_DMA call aids in the remapping of DMA mapped pages. This call will selectively suspend bus traffic while migrating the contents of DMA mapped pages. It also modifies the Translation Control Entries (TCEs) used for DMA accesses. Such functionality will allow for the removal of *difficult* memory sections on PPC64.

7.2 x86-64 Implementation

Although much of the memory hot-plug infrastructure discussed in this paper, such as the *sparsemem* implementation, is generic across all platforms, architecture specific support is still required due to the variance in memory management requirements for specific processor architectures. Fortunately, the changes to the x86-64 Linux kernel beyond *sparsemem* to support memory hotplug have been minimized to the following:

- Kernel Page Table Initialization (capacity addition)
- ZONE_NORMAL selection
- Kernel Page Table Tear Down (capacity reduction)

The x86-64 kernel doesn't require the HIGHMEM zone due to the large virtual address space provided by the architecture [28][12]. Thus, new memory regions discovered during memory hot-add operations result in expansion of the NORMAL zone. Conversely, because the x86-64 kernel only uses the DMA and NORMAL zones, removal of memory within each zone as discussed in 5 is required.

Much of the development of the kernel support for memory hotplug has relied on *logical* memory add and remove operations, which has enabled the use of existing platforms for prototyping. However, the x86-64 kernel has been tested and used on real hardware that supports memory hotplug. Specifically, the x86-64 memory hotplug kernels have been tested on a recently released Intel Xeon®[6] platform that supports physical memory hotplug operations.

[6]Xeon is a registered trademark of the Intel Corporation

One of the key pieces of supporting physical memory hotplug is notification of memory capacity changes from the hardware/firmware. The ACPI specification outlines basic information on memory devices that is used to convey these changes to the kernel. Accordingly, in order to fully support physical memory hotplug in the kernel the x86-64 kernel uses the ACPI memory hotplug driver to field notifications from firmware and notify the VM of the addition or reduction at runtime using the same interface employed by the logical operations. Further information on the ACPI memory hotplug driver support in the kernel may be found in [21].

7.3 IA64 Implementation

IA64 is one of architectures where Memory Hotplug is eagerly desired. From the view of Memory Hotplug, IA64 linux has following characteristics:

- The memory layout of IA64 is very sparse with lots of holes.
- For managing holes, `VIRTUAL_MEM_MAP` is used in some configurations.
- MAX_ORDER is not 11 but 18.
- IA64 supports a physical address bits of 50

Early lmbench2 data has shown that SPARSEMEM performs equivalently to DISCONTIGMEM+VIRTUAL_MEM_MAP. The data was taken on a non-NUMA machine. Further work should be done with other benchmarks and NUMA hardware.

7.3.1 SPARSEMEM and VIRTUAL MEM MAP

The VM uses a `memmap[]`, a linear array of page structures. With DISCONTIGMEM, `memmap[]` is divided into several `node_mem_maps`. In general, `memmap[]` is allocated in physically contiguous pages at boot time.

The memory layout of IA64 is very sparse with lots of holes. Sometimes there are GBs of memory holes, even for a non-NUMA machine. In IA64 DISCONTIGMEM, a `vmemmap` is used to avoid wasting memory. A `vmemmap` is a `memmap` which uses contiguous region of virtual address instead of contiguous physical memory. [7]

It is useful to hide holes and to create sparse `memmap[]`s. It resides in region 5 of the virtual address space, which uses virtual page table [8] like vmalloc.

Unfortunately, VIRTUAL_MEM_MAP is quite complicated. Because of the complications VIRTUAL_MEM_MAP presents, early designs for MEMORY_HOTPLUG were too complicated to be successfully implemented. SPARSEMEM cleanly removes VIRTUAL_MEM_MAP and thus avoids the associated complexity altogether. Because SPARSEMEM is simpler than VIRTUAL_MEM_MAP it is a logical replacement for VIRTUAL_MEM_MAP for situations other than just hotplug. SPARSEMEM divides the whole memmap into the section's `section_memmap`s. All `section_memmap`s reside in region 7 of the virtual address space. Region 7 is an identity mapped segment and handled by the fast TLB miss handler with big page size. If a hole covers the whole section, section_memmap is not allocated. Holes in a section are treated as reserved pages. For example, an HP rx2600 with 4GB of memory has the available physical memory at two locations with sizes of 1Gb and 3Gb. For VIRTUAL_MEM_MAP the holes would be represented by empty virtual space with vmemmap. SPARSEMEM handles a hole which covers an entire section with an invalid section.

[7] VIRTUAL_MEM_MAP is configurable independent of DISCONTIGMEM

[8] VHPT, Virtual Hash Page Table, is a hardware supported function to fill TLB

7.3.2 SPARSEMEM NUMA

The `mem_section[]` array is on the BP's node. Because `pfn_to_page()` accesses it, a non BP node `pfn_to_page()` is slightly more expensive. Besides boot time the section array is modified only during a hotplug event. These events should happen infrequently. This frequently accessed but rarely changing data suggests replicating the array into all nodes in order to eliminate the non BP node penalty. Hotplug memory updates would have to notify each node of modifications to the array.

7.3.3 Size of Section and MAX_ORDER

One feature which is very aggresive on IA64 is the configuration parameter FORCE_MAX_ZONEORDER. This overwrites MAX_ORDER to 18. For a 16kb page size the resultant MAX_ORDER region is 4Gb(18+14). This is done for supporting 4Gb HugetlbFS. SPARSEMEM constrains $PAGE\_SIZE * 2^{(MAX\_ORDER-1)}$ to be less than or equal to section size. For HugetlbFS we have: (1)the smallest size of section is 4GB and (2)holes smaller than 4GB consume reserved page structures. 18 of MAX_ORDER seems to be rather optimistic

value for Memory Hotplug. Currently, configuation of FORCE_MAX_ZONEORDER is modified at compile time. At configuration time, if HUGETLB isn't selected, FORCE_MAX_ZONEORDER can be configured to $11-20$. If HUGETLB is selected, MAX_ORDER and SECTION_SIZE are adjusted to support 4Gb HUGETLB Page.

7.3.4 Vast 50 Bits Address Space of IA64

The IA64 architecture supports a physical address bit limit of 50, which can addresss up to 1 petabyte of memory. A section array with a 256Mb section size requires 32Mb of data to cover the whole address range. The Linux kernel by default is configured to only use 44 bits maximum, which can address 16 terabytes of memory. This only requires 512Kb of data to cover the whole address range. The number of bits used is configurable at compile time.

8 Overlap with other efforts

During the development of memory hotplug the developers discovered two surprising things.

- Parts of the memory hotplug code were very useful to those who don't care at all about memory hotplug.
- Code others were developing without so much as a thought of memory hotplug proved useful for memory hotplug.

This section attempts to briefly mention these surprising overlaps with other independent development without straying too far from the topic of memory hotplug.

8.1 Memory Defragmentation

The primary concern for memory defragmentation within the VM subsystem is at the page level. At the heart of this concern is the page allocator and management of contiguous groups of pages. Memory requests can be made for for sizes in the range of a single page up to $2^{(MAX\_ORDER-1)}$ contiguous pages. As time goes by, various size allocations are obtained and freed. The page allocator attempts to intelligently group adjacent pages via the use of buddy allocator as previously described. However, it still may become difficult to satisfy requests for large size allocations. When a suitable size block is not found on the free lists, an attempt is made to reclaim pages so that a sufficiently large block can assembled. Unfortunately, not all pages can be reclaimed. For example, those in use for kernel data. The free area splitting concepts previously discussed address this issue. By grouping pages based on usage characteristics, the likelihood that a large block of pages can be reclaimed and ultimately allocated is greatly increased.

With memory hotplug, removing a memory section is somewhat analogous to allocating all the pages within the memory section. This is because all pages within the section must be free (not in use) before the section can be removed. Therefore, the concept of free list splitting can also be applied to memory sections for memory removal operations. Unfortunately however, memory sections do not map directly to memory blocks managed by the page allocator. Rather, a memory section consists of multiple contiguous $2^{(MAX\_ORDER-1)}$ page size blocks. The number of blocks is dependent on architecture specific SECTION_SIZE and MAX_ORDER definitions. Future work within the memory hotplug project is to extend the concepts used to avoid fragmentation to that of memory section size blocks. This will increase

the likelihood that memory sections can be removed.

8.2 NUMA Memory Management

In a NUMA system, memory hotplug must consider the case where all of the memory on a node might be added/removed. Structures to manage the node must be updated.

In addition, a user can specify nodes which are used by a user's tasks by using `mbind()` or `set_mempolicy()` in order to support load balancing among cpusets/dynamic partitioning. Memory hotplug has to not only update mempolicy information, but also make interfaces for load balancing scripts to move memory contents from nodes to other appropriate nodes.

8.2.1 Hotplug of Management Structures for a Node.

Structures which manage memory of a node must be updated in order to hotplug the node This section describes some of the structures.

pgdat To reduce expensive cross node memory accesses, Linux uses `pgdat` structures which include zone and zonelists. These structures are allocated on each node's local memory in order to reduce access costs. If a new node is hotplug added, its `pgdat` structure should be allocated on its own node. Normally, there are no mm structures for the node until the pgdat is initialized, so pgdat has to be allocated by special routine early in the boot process. This allocation (getting a virtual address and mapping physical address to it) is like a `ioremap()`, but it should be mapped on cached area unlike `ioremap()`.

zonelist The zonelist is an array of zone addresses, and it is ordered by which zone should be used for its node. Its order is determined by access cost from a cpu to memory and the zone's attributes. This implies when a node with memory is hotplugged, all the node's zonelists which are being accessed must be updated. For updating, the options are:

- getting locks
- giving up reordering
- stop other cpus while updating

Stopping other cpus while updating may be the best way, because there is no impact on performance of page allocation unless a hotplug event is in progress. In addition, more structures than just zonelists need updating. For example, mempolicies of each process have to be updated to avoid using a removed node. To update them, the system has to remember all of the processes' mempolicies. Linux does not currently do this, so further development is necessary.

8.2.2 Scattered Clone Structures Among Nodes

Pgdat, which includes zone and zonelist, is used to manage its own node, but some of data structures' clones are allocated on each of the nodes for light weight accesses. One current example is `NODE_DATA()` on IA64 implementation. `NODE_DATA(node_id)` macro points to each `node_id`'s pgdat. In the IA64 implementation, `NODE_DATA(node_id)` is not just an array like it is in the IA32 implementation. This data is localized on each node and it can be obtained from `per_cpu` data. In this case, all of the nodes have clones of `pg_data_ptrs[]` array.

```
#define  local_node_data                    \
           local_cpu_data->node_data
#define  NODE_DATA(nid)                     \
           local_node_data->pg_data_ptrs[nid]
```

Besides `NODE_DATA()`, many other data structures which are often accessed are localized to each node. This implies that all of the node copies must also be updated when a hotplug event occurs. To update them, `stop_machine_run` may prove to be the best method of serializing access.

8.2.3 Process Migration on NUMA

It is important to determine what the best destination node is for migration of memory contents. This applies not only automatic migration, but also "manual page migration" as proposed by Ray Bryant at SGI. With manual page migration a load balancer script can specify the destination node for migrating existing memory. A potential interface would be a simple system call like `sys_migrate_pages(pid, oldnode, newnode)`.

However, if there are too many nodes (ex, 128 nodes) and tasks (ex, 256 processes) in the system, this system call will be called too frequently. Therefore, Ray Bryant is proposing an array interface to specify each node to avoid too many calls: `sys_migrate_pages(pid, count, old_nodes, new_nodes)`.

The arguments to `sys_migrate_pages()` `old_nodes` and `new_nodes` are the sets of source and destination nodes and count is the number of elements in each array. Therefore, a user can just call `sys_migrate_pages()` once for each task. If each task uses shared message blocks, there will be a large reduction in the number of system calls.

9 Conclusion

Hotplug Memory is real and achievable due to the dedication of its developers. This paper has shown that the issues with memory hotplug have been well thought out. Most of memory hotplug has already been implemented and is maintained in the `-mhp` tree[3]—broken down into the smallest possible independent pieces for continued development. These small pieces are released early and often. As individual pieces of this code become ready for consumption by the general public they are merged upstream. By maintaining this separate tree which is updated at least once per -rc release, hotplug developers have been able to test and stabilize an increasing amount of memory hotplug code. Thus, the pieces that get merged upstream are small, non-disruptive, and well tested.

Memory hotplug is a model of how a large, disruptive feature can be developed and merged into a continuously stable kernel. In fact, having a stable kernel has made development much more disciplined, debugging easier, conflicts with other developers easier to identify, feedback more thorough, and generally has been a blessing in disguise.

If in a parallel universe somewhere Andrew Morton gave his keynote today instead of a year ago I suspect he would say something different. The parallel universe Andrew Morton might say:

"Some features tend to be pervasive and have their little sticky fingers into lots of different places in the code base. An example of which comes to mind is CPU hot plug, and memory hot unplug. We may not be able to accept these features into a 2.7 development kernel due to their long-term impact on stabilizing that kernel. To make it easier on the developers of features like these we have decided to never have a

2.7 development kernel. Because of this, I expect CPU hot plug and memory hot unplug to be merged with relative ease as they reach the level of stability of the rest of the kernel."

10 Acknowledgments

Special thanks to Ray Bryant of SGI for his review on NUMA migration, New Energy and Industrial Technology Development Organization for funding some of the contributions from the authors who work at Fujitsu, Martin Bligh for his NUMA work and his work on IBMs test environment, Andy Whitcroft for his work on SPARSEMEM, Andrew Morton and the quilt developers for giving us something to manage all of these patches, Sourceforge for hosting our mailing list, OSDL for the work of the Hotplug SIG—especially their work on testing, and Martine Silbermann for valuable feedback.

11 Legal Statement

References

[1] `http://www.groklaw.net/article.php?story=20040802115731932`

[2] M. Tosatti (`marcelo.tosatti@cyclades.com`) (14 Oct 2004), *Patch: Migration Cache*. Email to Dave Hansen, Iwamoto Toshihiro, Hiroyuki Kamezawa, `linux-mm@kvack.org` (`http://lwn.net/Articles/106977/`)

[3] `http://sr71.net/patches/`

[4] C. Lameter (clameter@sgi.com) (21 Oct 2004) *Patch: Hugepages demand paging V1[0/4]: Discussion and Overview*. Email to Kenneth Chen, William Lee Irwin III, Ray Bryant, `linux-kernel@vger.kernel.org` (`http://lwn.net/Articles/107719/`)

[5] H. Takahashi, 2004 [online]. Linux memory hotplug for Hugepages. Available from: `http://people.valinux.co.jp/~taka/hpageremap.html` [Accessed 2004].

[6] D. Hansen, M. Kravetz, B. Christiansen, M. Tolentino. Hotplug Memory and the Linux VM. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Ontario, Canada, pages 278–294, July 2004.

[7] K. Knowlton. A Fast Storage Allocator. In *Communications of the ACM*, Vol. 8, Issue 10, pages 623–624, October 1965.

[8] M. Gorman. Understanding the Linux Virtual Memory Manager, Prentice Hall, NJ, 2004.

[9] W. Bolosky, R. Fitzgerald, M. Scott. Simple But Effective Techniques for

NUMA Memory Management. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 19–31, 1989.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating System Principles*, pages 164–177, October 2003.

[11] E. Demaine, J. Munro. Fast Allocation and Deallocation with an Improved Buddy System. In *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 84–96, 1999.

[12] Intel Corporation. 64-Bit Extension Technology Software Developer's Guide. 2004.

[13] K. Li and K. Peterson. Evaluation of Memory System Extensions. In *Proceedings of 18th Annual International Symposium on Computer Architecture*, pages 84–93, 1991.

[14] D. Mosberger, S. Eranian. ia-64 Linux Kernel Design and Implementation. Prentice Hall, NJ, 2002.

[15] Z. Mwaikambo, A. Raj, R. Russell, J. Schopp, S. Vaddagiri. Linux Kernel Hotpug CPU Support. In *Proceedings of the Ottawa Linux Symposium*, pages 467–479, July 2004.

[16] J. Peterson, T. Norman. Buddy Systems. In *Communications of the ACM*, Vol. 20, Issue 6, pages 421–431, June 1977.

[17] C. Waldspurger. Memory Resource Management in VMware ESX Server. In *Procedings of the 5th Symposium on Operating System Design and Implementation*, pages 181–194, 2002.

[18] M.S. Johnstone, P.R. Wilson. The Memory Fragmentation Problem: Solved? In *International Symposium on Memory Management*, pages 26–36, Vancouver, British Columbia, Canada, 1998.

[19] Linux Weekly News, 2005 [online]. Yet another approach to memory fragmentation. Available from `http://lwn.net/121618/` [Accessed Feb. 2005]

[20] ACPI Specification Version 3.0 `http://www.acpi.info/spec.htm`

[21] L. Brown, *et al*. The State of ACPI in the Linux Kernel. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Ontario, Canada, July 2005.

[22] B. Jacob, T. Mudge. Virtual Memory in Contemporary Microprocessors, IEEE Micro, pages 60–75, July 1998

[23] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, J. Chew. Machine Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures, In *Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 1987.

[24] S. Hand. Self-Paging in the Nemesis Operating System, In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 73–86, February 1999.

[25] P. Denning. Virtual Memory, In *ACM Computing Surveys (CSUR)*, Vol. 2, Issue 3, pages 153–189, September 1970.

[26] A. Bensoussan, C. Clingen, R. Daley. The Multics Virtual Memory, In *Communications of the ACM*, Vol. 15, Issue 5, pages 308–318, May 1972.

[27] V. Abrossimov, M. Rozier. Generic Virtual Memory Management for Operating System Kernels. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 123–136, November 1989.

[28] A. Kleen. Porting Linux to x86-64. In *Proceedings of the Ottawa Linux Symposium,*, Ottawa, Ontario, Canada, July 2001.

[29] IBM. Dynamic Logical Partitioning in IBM eserver pSeries. October 2002.

[30] D.H. Brown Associates, Inc. Capacity on Demand A Requirement for the e-Business Environment. IBM White Paper. September 2003.

[31] L.D. Paulson. Computer System, Heal Thyself. In *IEEE Computer*, Vol. 35, Issue 8, August 2002.

Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*